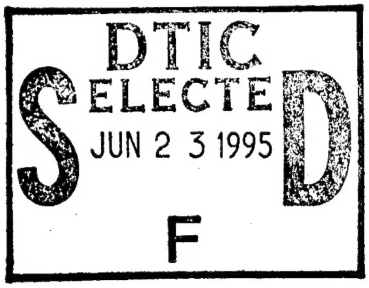
REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 11, 1995	Final

4. TITLE AND SUBTITLE:
Ada Compiler Validation Summary Report, VC# 950511W1.11382
Texas Instruments, Incorporated -- Compiler Name: F-16 Modular Mission Computer Ada Compilation System, Version 2_5_01

5. FUNDING NUMBERS



6. AUTHOR(S)

Software Standards Validation Group

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility
Language Control Facility, 645 C-CSG/SCSL
Area B, Building 676
Wright-Patterson AFB, OH 45433-6503

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office, Defense Information System Agency
Code JEXEV, 701 S. Courthouse Rd., Arlington, VA
22204-2199

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; Distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

This Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on May 11, 1995.
Host Computer System: VAXstation 4000/90 under VAX/VMS, Version 5.5-2H4
Target Computer System: F-16 Modular Mission Computer (bare machine)

14. SUBJECT TERMS
Ada Programming Language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES
48

16. PRICE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED

NSN 7540-01-280-5500

AVF Control Number: AVF-VSR-603.0495
Date VSR Completed: 22 MAY 95
95-01-03-TXI

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 950511W1.11382
Texas Instruments, Incorporated
F-16 Modular Mission Computer Ada Compilation System, Version 2_5_01
VAXstation 4000/90 under VAX/VMS, V5.5-2H4 =>
F-16 Modular Mission Computer (Bare Machine)

(Final)

Prepared By:
Ada Validation Facility
88 CG/SCTL
Wright-Patterson AFB OH 45433-5707

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 11 MAY 95.

Compiler Name and Version: F-16 Modular Mission Computer
Ada Compilation System, Version 2_5_01

Host Computer System: VAXstation 4000/90
under VAX/VMS, V5.5-2H4

Target Computer System: F-16 Modular Mission Computer
(Bare Machine)

Customer Agreement Number: 95-01-03-TXI

See section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 950511W1.11382 is awarded to Texas Instruments, Incorporated. This certificate expires on March 31, 1998.

This report has been reviewed and is approved.

Brian P. Andrews
Ada Validation Facility
Brian P. Andrews
Technical Director
88 CG/SCTL
Wright-Patterson AFB OH 45433-5707

Ada Validation Organization
Director, Computer and Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Ada Joint Program Office
Donald J. Reifer
Director, AJPO
Defense Information Systems Agency,
Center for Information Management

Accesion For	
NTIS CRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and / or Special
A-1	

DECLARATION OF CONFORMANCE

Customer: Texas Instruments, Incorporated

Ada Validation Facility: ATTN: Brian P. Andrews
88 CG/SCSB
3810 Communications Blvd., Suite 1
Wright-Patterson AFB OH 45433-5706

ACVC Version: 1.11

Ada Implementation:

Compiler Name and Version: F-16 Modular Mission Computer Ada Compilation System, Version 2_5_01

Host Computer System: VAXstation 4000/90 running VAX/VMS V5.5-2H4

Customer's Declaration

I, the undersigned, representing Texas Instruments, Incorporated, declare that Texas Instruments, Incorporated has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration.

Stewart L French Date: 3/23/95

Stewart French
Member, Group Technical Staff
Texas Instruments, Incorporated
PO Box 659305, M/S 8496
Plano, TX 75086

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro95] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro95]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Computer and Software Engineering Division
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311-1772

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro95] Ada Compiler Validation Procedures, Version 4.0, Ada Joint Program Office, January 1995.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1), and possibly removing some inapplicable tests (see section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process, or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro95].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 22 November 1993.

B27005A	E28005C	B28006C	C32203A	C34006D	C35507K
C35507L	C35507N	C35507O	C35507P	C35508I	C35508J
C35508M	C35508N	C35702A	C35702B	C37310A	B41308B
C43004A	C45114A	C45346A	C45612A	C45612B	C45612C
C45651A	C46022A	B49008A	B49008B	A54B02A	C55B06A
A74006A	C74308A	B83022B	B83022H	B83025B	B83025D
C83026A	B83026B	C83041A	B85001L	C86001F	C94021A
C97116A	C98003B	BA2011A	CB7001A	CB7001B	CB7004A
CC1223A	BC1226A	CC1226B	BC3009B	BD1B02B	BD1B06A
AD1B08A	BD2A02A	CD2A21E	CD2A23E	CD2A32A	CD2A41A
CD2A41E	CD2A87A	CD2B15C	BD3006A	BD4008A	CD4022A
CD4022D	CD4024B	CD4024C	CD4024D	CD4031A	CD4051D
CD5111A	CD7004C	ED7005D	CD7005E	AD7006A	CD7006E
AD7201A	AD7201E	CD7204B	AD7206A	BD8002A	BD8004C
CD9005A	CD9005B	CDA201E	CE2107I	CE2117A	CE2117B
CE2119B	CE2205B	CE2405A	CE3111C	CE3116A	CE3118A
CE3411B	CE3412B	CE3607B	CE3607C	CE3607D	CE3812A
CE3814A	CE3902B				

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by the ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 20 tests check for the predefined type LONG_INTEGER; for this implementation, there is no such type:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45613C	C45614C	C45631C	C45632C	B52004D
C55B07A	B55B09C	B86001W	C86006C	CD7101F

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT; for this implementation, there is no such type.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45624A..B (2 tests) check that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types and the results of various floating-point operations lie outside the range of the base type; for this implementation, MACHINE_OVERFLOWS is TRUE.

B86001Y uses the name of a predefined fixed-point type other than type DURATION; for this implementation, there is no such type.

C96005B uses values of type DURATION's base type that are outside the range of type DURATION; for this implementation, the ranges are the same.

CD1009C checks whether a length clause can specify a non-default size for a floating-point type; this implementation does not support such sizes.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use length clauses to specify non-default sizes for access types; this implementation does not support such sizes.

The following 264 tests check operations on sequential, text, and direct access files; this implementation does not support external files:

CE2102A..C (3)	CE2102G..H (2)	CE2102K	CE2102N..Y (12)
CE2103C..D (2)	CE2104A..D (4)	CE2105A..B (2)	CE2106A..B (2)
CE2107A..H (8)	CE2107L	CE2108A..H (8)	CE2109A..C (3)

CE2110A..D (4)	CE2111A..I (9)	CE2115A..B (2)	CE2120A..B (2)
CE2201A..C (3)	EE2201D..E (2)	CE2201F..N (9)	CE2203A
CE2204A..D (4)	CE2205A	CE2206A	CE2208B
CE2401A..C (3)	EE2401D	CE2401E..F (2)	EE2401G
CE2401H..L (5)	CE2403A	CE2404A..B (2)	CE2405B
CE2406A	CE2407A..B (2)	CE2408A..B (2)	CE2409A..B (2)
CE2410A..B (2)	CE2411A	CE3102A..C (3)	CE3102F..H (3)
CE3102J..K (2)	CE3103A	CE3104A..C (3)	CE3106A..B (2)
CE3107B	CE3108A..B (2)	CE3109A	CE3110A
CE3111A..B (2)	CE3111D..E (2)	CE3112A..D (4)	CE3114A..B (2)
CE3115A	CE3119A	EE3203A	EE3204A
CE3207A	CE3208A	CE3301A	EE3301B
CE3302A	CE3304A	CE3305A	CE3401A
CE3402A	EE3402B	CE3402C..D (2)	CE3403A..C (3)
CE3403E..F (2)	CE3404B..D (3)	CE3405A	EE3405B
CE3405C..D (2)	CE3406A..D (4)	CE3407A..C (3)	CE3408A..C (3)
CE3409A	CE3409C..E (3)	EE3409F	CE3410A
CE3410C..E (3)	EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A..C (3)	CE3414A
CE3602A..D (4)	CE3603A	CE3604A..B (2)	CE3605A..E (5)
CE3606A..B (2)	CE3704A..F (6)	CE3704M..O (3)	CE3705A..E (5)
CE3706D	CE3706F..G (2)	CE3804A..P (16)	CE3805A..B (2)
CE3806A..B (2)	CE3806D..E (2)	CE3806G..H (2)	CE3904A..B (2)
CE3905A..C (3)	CE3905L	CE3906A..C (3)	CE3906E..F (2)

CE2103A, CE2103B, and CE3107A use an illegal file name in an attempt to create a file and expect NAME_ERROR to be raised; this implementation does not support external files and so raises USE_ERROR. (See section 2.3.)

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 24 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B24009A	B33301B	B38003A	B38003B	B38008B	B38009B
B85008G	B85008H	BC1303F	BC3005B	BD2B03A	BD2D03A
BD4003A					

CD1009A, CD1009I, CD1C03A, CD2A22J, CD2A31A..C (3 tests) were graded passed by Evaluation Modification as directed by the AVO. These tests use instantiations of the support procedure LENGTH_CHECK, which uses Unchecked_Conversion according to the interpretation given in AI-00590. The AVO ruled that this interpretation is not binding under ACVC 1.11; the tests are ruled to be passed if they produce Failed messages only from the instances of LENGTH_CHECK--i.e, the allowed Report.Failed messages have the general form:

" * CHECK ON REPRESENTATION FOR <TYPE_ID> FAILED."

AD9001B was graded passed by Test Modification as directed by the AVO. This test checks that no bodies are required for interfaced subprograms; among the procedures that is used is one with a parameter of mode OUT (line 36). This implementation does not support pragma INTERFACE for procedures with parameters of mode OUT. The test was modified by commenting out line 36 and 40; the modified test was passed.

CE2103A, CE2103B, and CE3107A were graded inapplicable by Evaluation Modification as directed by the AVO. The tests abort with an unhandled exception when USE_ERROR is raised on the attempt to create an external file. This is acceptable behavior because this implementation does not support external files (cf. AI-00332).

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For technical and sales information about this Ada implementation, contact:

Stewart French
Texas Instruments, Incorporated
PO Box 659305, M/S 8496
Plano, TX 75086
(214) 575-4272

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro95].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

The list of items below gives the number of ACVC tests in various categories. All tests were processed, except those that were withdrawn because of test errors (item b; see section 2.1), those that require a floating-point precision that exceeds the implementation's maximum precision (item e; see section 2.2), and those that depend on the support of a file system -- if none is supported (item d). All tests passed, except those that are listed in sections 2.1 and 2.2 (counted in items b and f, below).

a) Total Number of Applicable Tests	3558	
b) Total Number of Withdrawn Tests	104	
c) Processed Inapplicable Tests	43	
d) Non-Processed I/O Tests	264	
e) Non-Processed Floating-Point Precision Tests	201	
f) Total Number of Inapplicable Tests	508	(c+d+e)
g) Total Number of Tests for ACVC 1.11	4170	(a+b+f)

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded directly onto the host computer.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

The tests were compiled and linked on the host computer system, as appropriate. The executable images were transferred to the target computer system by the IEEE-488, and run. The results were captured on the host computer system.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. It also indicates the default options. No explicit options were used for testing this implementation.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$MAX_IN_LEN	499 -- Value of V
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"

```
$MAX_STRING_LITERAL    '"' & (1..V-2 => 'A') & '"'
```

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2_147_483_647
$DEFAULT_MEM_SIZE	16_777_216
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	MACS
$DELTA_DOC	0.000000000465661287307739257812 5
$ENTRY_ADDRESS	SYSTEM."+"(16)
$ENTRY_ADDRESS1	SYSTEM."+"(17)
$ENTRY_ADDRESS2	SYSTEM."+"(2)
$FIELD_LAST	2_147_483_647
$FILE_TERMINATOR	' '
$FIXED_NAME	NO_SUCH_TYPE
$FLOAT_NAME	NO_SUCH_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	100_000.0
$GREATER_THAN_DURATION_BASE_LAST	10_000_000.0
$GREATER_THAN_FLOAT_BASE_LAST	1.8E+308
$GREATER_THAN_FLOAT_SAFE_LARGE	5.0E307

```
$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE
                        9.0E37

$HIGH_PRIORITY          99

$ILLEGAL_EXTERNAL_FILE_NAME1
                        "/illegal/file_name/2}]$%FILE1.DAT"

$ILLEGAL_EXTERNAL_FILE_NAME2
                        "/illegal/file_name/2}]$%FILE2.DAT"

$INAPPROPRIATE_LINE_LENGTH
                        -1

$INAPPROPRIATE_PAGE_LENGTH
                        -1

$INCLUDE_PRAGMA1        PRAGMA INCLUDE ("A28006D1.TST")

$INCLUDE_PRAGMA2        PRAGMA INCLUDE ("B28006D1.TST")

$INTEGER_FIRST          -2147483648

$INTEGER_LAST           2147483647

$INTEGER_LAST_PLUS_1    2147483648

$INTERFACE_LANGUAGE     C

$LESS_THAN_DURATION     -100_000.0

$LESS_THAN_DURATION_BASE_FIRST
                        -10_000_000.0

$LINE_TERMINATOR        ASCII.LF

$LOW_PRIORITY           0

$MACHINE_CODE_STATEMENT
                        CODE_0'(OP => NOP);

$MACHINE_CODE_TYPE      CODE_0

$MANTISSA_DOC           31

$MAX_DIGITS             15

$MAX_INT                2147483647

$MAX_INT_PLUS_1         2147483648

$MIN_INT                -2147483648

$NAME                   TINY_INTEGER
```

MACRO PARAMETERS

$NAME_LIST	MACS
$NAME_SPECIFICATION1	GEORGE$DUA1:[VALIDATION.91-03-18-VRX.TESTS]X2120A
$NAME_SPECIFICATION2	GEORGE$DUA1:[VALIDATION.91-03-18-VRX.TESTS]X2120B
$NAME_SPECIFICATION3	GEORGE$DUA1:[VALIDATION.91-03-18-VRX.TESTS]X3119A
$NEG_BASED_INT	16#F000000E#
$NEW_MEM_SIZE	16_777_216
$NEW_STOR_UNIT	8
$NEW_SYS_NAME	MACS
$PAGE_TERMINATOR	ASCII.LF & ASCII.FF
$RECORD_DEFINITION	RECORD SUBP: OPERAND; END RECORD;
$RECORD_NAME	CODE_0
$TASK_SIZE	32
$TASK_STORAGE_SIZE	1024
$TICK	0.01
$VARIABLE_ADDRESS	VAR_1'ADDRESS
$VARIABLE_ADDRESS1	VAR_2'ADDRESS
$VARIABLE_ADDRESS2	VAR_3'ADDRESS
$YOUR_PRAGMA	PASSIVE

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

VADS ADA — invoke the Ada compiler

Syntax

```
VADS ADA [qualifiers] [source_file]... [object_file]...
```

Arguments

object_file Non–Ada object filenames. These files are passed to the linker and are linked with the specified Ada object files.

qualifiers Qualifiers to the compiler. These are:

/APPEND Must be used with **/OUTPUT**. It Appends output to ***filename***.

/DEBUG Write out the **GNRX.LIB** file in ASCII.

/DEFINE=("identifier:type=value", ...)
Define identifier of a specified type and value.

/DEPENDENCIES Analyze for dependencies only; no link is performed if this qualifier is given (**/MAIN** and **/OUTPUT** qualifiers must not be used with this qualifier).

/ERRORS[=(*option* [, ...])]
Process compilation error messages using the ERROR tool and direct the output to **SYS$OUTPUT**; the parentheses can be omitted if only one qualifier is given (by default, only lines containing errors are listed).

Options:

LISTING	List entire input file.
EDITOR[=*"editor"*]	Insert error messages into the source file and call a text editor (**EDT** by default). To use an editor other than **EDT**, specify it as the quoted string ***"editor"***.
OUTPUT[=*filename*]	Direct error processed output to the specified filename; if no filename is given, the source filename is used with a file extension .ERR.
BRIEF	list only the affected lines [default]

Use only one of the **BRIEF, LISTING, OUTPUT** or **EDITOR** options in a single command.

/EXECUTABLE=*filename*
Provide an explicit name for the executable when used with the /MAIN qualifier; the ***filename*** value must be supplied (if the file type is omitted, **.VOX** is assumed).

/FILE_LIST=*filename* Compile the source files listed in ***filename***. When **/FILE_LIST=*filename*** is used, ***source_file*** is not required.

/FULL_DIANA Do not trim the DIANA tree before output to net files. To save disk space, the DIANA tree is trimmed so that all pointers to nodes that did not involve a subtree that define a symbol table are nulled (unless those nodes are part of the body of an inline or generic or certain other values that are retained for the debugging or compilation information). The trimming generally removes initial values of variables and all statements.

/GVAS_SUGGESTED
Display suggested values for MIN_GVAS_ADDR and MAX_GVAS_ADDR INFO directives.

/KEEP_IL Keep the intermediate language (IL) file produced by the compiler front end. The IL file is placed in the **OBJECTS** directory, with the name ***ADA_SOURCE*.I**.

/LIBRARY=*library_name*
Operate in VADS library ***library_name*** (the current working directory is the default).

/LINK_ARGUMENTS=*"value"*
Pass command qualifiers and parameters to the linker.

/MAIN[=*unit_name*] Produce an executable program using the named unit as the main program; if no value is given, the name is derived from the first Ada filename parameter (the **.A** suffix is removed); the executable filename is derived from the main program name unless the **/EXECUTABLE** qualifier is used.

/NO_CODE_SHARING
Compile all generic instantiations without sharing code for their bodies. This option overrides the SHARE_BODY INFO directive and the SHARE_CODE or SHARE_BODY pragmas.

/NOCONTROL Suppress "control" messages emitted when **pragma** PAGE and/or **pragma** LIST are encountered.

/NOOPTIMIZE Do not optimize.

/OPTIMIZE[=*number*]
Invoke the code optimizer. An optional digit provides the level of optimization. **/OPTIMIZE=4** is the default.

/OPTIMIZE	no digit, full optimization
/OPTIMIZE=0	no optimization
/OPTIMIZE=1	copy propagation, CONST folding, removing dean variables, subsuming moves between scalar variables
/OPTIMIZE=2	add common subexpression elimination within basic blocks
/OPTIMIZE=3	add global common subexpression elimination
/OPTIMIZE=4	add hoisting invariants from loops and address optimizations
/OPTIMIZE=5	add range optimizations and one pass of reducing induction expressions
/OPTIMIZE=6	add unrolling of inner–most loops
/OPTIMIZE=7	add one more pass of induction expression reduction
/OPTIMIZE=8	add one more pass of induction expression reduction
/OPTIMIZE=9	add one more pass of induction expression reduction and add hoisting expressions common to the **then** and the **else** parts of if statements

/OUTPUT=*filename* Direct the output to ***filename*** (the default is **SYS$OUTPUT**).

/PRE_PROCESS Invoke the Ada Preprocessor, **VADS APP**

/RECOMPILE_LIBRARY=*VADS_library*
Force analysis of all generic instantiations causing reinstantiation of any that are out of date.

/RECREATE_GVAS Reinitialize the library's GVAS and the GVAS_TABLE file and exit. No compilations are performed.

/SHOW	Show the name of the tool executable but do not execute it.
/SHOW_ALL	Print the name of the front end, code generator, optimizer, linker and list the tools that are invoked.
/SUPPRESS	Apply **pragma** SUPPRESS for all checks to the entire compilation
/TIMING	Print timing information for the compilation.
/VERBOSE	Print information for the compilation.
/WARNINGS **/NOWARNINGS**	Specify display of warning diagnostics [Default: **/WARNINGS**].
source_file	Name of the source file to compile.

Description

The command **VADS ADA** executes the Ada compiler and compiles the named Ada source file. The file must reside in a VADS library directory. The **ADA.LIB** file in this directory is modified after each Ada unit is compiled.

By default, **VADS ADA** produces only object and net files. If the **/MAIN** qualifier is used, the compiler automatically invokes **VADS LD** and builds a complete program with the named library unit as the main program.

The compiler generates object files in VOX format.

Non-Ada object files can be given as arguments to **VADS ADA**. These files are passed on to the linker and are linked with the specified Ada object files.

Command line qualifiers can be specified in any order but the order of compilation and the order of the files passed to the linker is significant.

Several VADS compilers may be simultaneously available on a single system. The **VADS ADA** command within any version of VADS on a system executes the correct compiler components based upon visible library directives.

Program listings with a disassembly of machine code instructions are generated by **VADS DB** or **VADS DAS**.

Diagnostics

The diagnostics produced by the VADS compiler are self-explanatory. Most refer to the RM. Each RM reference includes a section number and optionally, a paragraph number enclosed in parentheses.

REFERENCES: APP Prog—9 , **VADS APP** Ref—30 **/ERRORS** User—42,. Optimization User—40, **pragma** OPTIMIZE_CODE(OFF) Prog—82, .**pragma** SUPPRESS(ALL_CHECKSProg—84, **VOX** format Prog—65, **VADS DAS** Ref—37, **VADS DB** Ref—39, **VADS ERROR** Ref—41, **VADS LD** Ref—53, **VADS MKLIB** Ref—62

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

VADS LD — prelinker

Syntax

```
VADS LD [qualifiers] unit_name [linker_options]
```

Arguments

linker_options All arguments after ***unit_name*** are passed on to the linker. These arguments can be linker options, names of object files or archive libraries or library abbreviations.

qualifiers Qualifiers to the **VADS LD** command. These are:

/APPEND Must be used with **/OUTPUT**. It Appends output to ***filename***.

/EARLY="*unit_name*"
Force the given unit to elaborate as early as possible (***unit_name*** must be enclosed in double quotes).

/EXECUTABLE[=*filename*]
Put the output in the named file. The default executable names are **<*main_unit*>.EXE** on self hosts or **<*main_unit*>.VOX** on cross targets.

/FILES Print a list of dependent files in elaboration order and suppress linking.

/LIBRARY=*library_name*
Collect information required for linking in ***library_name*** instead of the current directory. However, place the executable in the current directory.

/LINK_ARGS_FILE=*filename*
Read list of options and/or object files to be passed to the VMS linker from file ***filename***. This option is used when the number or length of the arguments is so large that it cannot fit on the VMS command line. In ***filename***, the arguments can be listed one per line or there can be multiple arguments on one line. Any number of spaces and/or blank lines can delimit each argument. This option can be used in conjunction with **/LINK_OPTIONS**.

/LINK_OPTIONS="*object_file_or_qualifier*[,...]"
Add the options surrounded by quotes to the invocation of the linker.

/OUTPUT=*filename* Direct output to ***filename***. Default is **SYS$OUTPUT**.

/SHOW Show the name of the tool executable but do not execute it.

/TABLE List the symbols in the elaboration table to standard output.

/UNITS Print a list of dependent units in order and suppress linking.

/VERBOSE Generate an options file, usable by the cross linker, that has the name of the executable but with the extension **.OPT**.

/VERIFY Print the VMS linker command but suppress execution.

/WARNINGS Suppress warning messages.

unit_name Name of an Ada unit.

Description

VADS LD collects the object files needed to make ***unit_name*** a main program and calls the xlink linker to link together all Ada and other language objects required to produce an executable. ***unit_name*** must be a non–generic subprogram that is either a procedure or a function that returns an Ada STANDARD.INTEGER (the predefined **type** INTEGER). The utility uses the net files produced by the Ada compiler to check dependency information. **VADS LD** produces an exception mapping table, a unit elaboration table and passes this information to the linker. The

elaboration list generated by **VADS LD** does not include library level packages that do not need elaboration. Similarly, packages that contain no code that raises an exception no longer have exception tables.

VADS LD reads instructions for generating executables from the **ADA.LIB** file in the VADS libraries on the search list. Besides information generated by the compiler, these directives include WITH*n* directives automatically link object modules compiled from other languages or Ada object modules not named in context clauses in the Ada source. Any number of WITH directives can be placed in a library but they must be numbered contiguously beginning at WITH1. The directives are recorded in the library's **ADA.LIB** file and have the following form:

```
WITH1|LINK|object_file|
WITH2|LINK|archive_file|
```

WITH directives can be placed in the local Ada libraries or in any VADS library on the search list.

A WITH directive in a local VADS library or earlier on the library search list hides the same numbered WITH directive in a library later in the library search list.

Use **VADS INFO** to change or report library directives in the current library.

Diagnostics

Self–explanatory diagnostics are produced for missing files, etc. Occasional additional messages are produced by the linker.

Files

Normally, **VADS LD** generates an intermediate file with the process ID as a substring, VADSOPTION<***process_ID***>.**OPT**.

With either the **/VERIFY** or **/VERBOSE** qualifiers, however, **VADS LD** produces the intermediate file, <***main_unit***>.**OPT**.

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is
   ..........

   type INTEGER is range -2147483648 .. 2147483647;

   type SHORT_INTEGER is range -32768 .. 32767;

   type TINY_INTEGER is range -128 .. 127;

   type LONG_FLOAT is digits 15 range
      -1.79769313486232E+308 .. +1.79769313486232E+308;

   type FLOAT is digits 6 range
      -3.40282E+38 .. 3.40282E+38;

   type SHORT_FLOAT is digits 6 range
      -3.40282E+38 .. 3.40282E+38;

   type DURATION is delta 0.0001 range
      -214748.3648 .. 214748.3647;

   ..........
end STANDARD;
```

Appendix F

Implementation Dependencies

This chapter defines the implementation-dependent characteristics of the RTE as required by MIL-STD-1815A, Appendix F. The VADS compiler provides these features:

- shared generic bodies
- all-Ada run time system
- representation clauses to the bit level and pragma PACK (Ada RM 13.1)
- length clauses and unsigned types (8- and 16-bit) (Ada RM 13.2)
- enumeration representation clauses (Ada RM 13.3)
- record representation clauses (Ada RM 13.4)
- interrupt entries (Ada RM 13.5.1)
- representation attributes (Ada RM 13.7.2)
- machine code insertions and pragma IMPLICIT_CODE (Ada RM 13.8)
- interface programming features, including pragma INTERFACE, pragma EXTERNAL_NAME, pragma EXTERNAL, pragma INTERFACE_NAME, WITH directives, VADS INFO, and external dependencies capabilities (Ada RM 13.9)
- unchecked deallocations (Ada RM 13.10.1)
- unchecked conversions (Ada RM 13.10.2)
- pool-based memory allocation option

F.1 Implementation-Dependent PRAGMAs

Each of this implementation's pragmas is briefly described here. Additional information on some of these pragmas is found under discussions of particular language constructs.

F.1.1 PRAGMA alignment

PRAGMA alignment(object, power_of_2_byte_alignment) (VADScross only)

allows the user to specify alignment for objects declared in a package specification or body.

F.1.2 PRAGMA built_in

PRAGMA built_in

may be used in some parts of the code for TEXT_IO, MACHINE_CODE, UNCHECKED_CONVERSION, UNCHECKED_DEALLOCATION and lower level support packages in STANDARD. It is reserved and cannot be accessed directly.

F.1.3 PRAGMA controlled

PRAGMA controlled

is recognized by the implementation but has no effect in the current release.

F.1.4 PRAGMA elaborate

PRAGMA elaborate

is implemented as described in Appendix B of the Ada RM.

F.1.5 PRAGMA external

PRAGMA external(language, subprogram)

supports calling Ada subprograms from foreign languages. The compiler generates code for the subprogram that is compatible with the calling conventions of the foreign language. The subprogram may also be called from Ada normally. The supported languages and restrictions on parameter and result types are the same as for pragma INTERFACE. This pragma only has an effect when the calling conventions of the foreign language differ from those of Ada.

F.1.6 PRAGMA external_name

PRAGMA external_name(subprogram, link_name)

allows the user to specify a link for an Ada variable or subprogram so that the object can be referenced from other languages. The PRAGMA is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification.

Objects must be variables defined in a package specification; subprograms can be either library level or within a package specification.

This pragma is allowed with inline subprograms but disallowed with inline_only subprograms. It also cannot be used on objects created by renaming declarations.

F.1.7 PRAGMA implicit_code

PRAGMA implicit_code

The IMPLICIT_CODE pragma specifies that implicit code generated by the compiler is allowed (ON) or disallowed (OFF) and is used only within the declarative part of a machine code procedure. Implicit code includes preamble and trailer code (e.g., code used to move parameters from and to the stack). Use of PRAGMA implicit_code does not eliminate code generated for run time checks nor does it eliminate call/return instructions. These can be eliminated by PRAGMA suppress and PRAGMA inline, respectively. A warning is issued if OFF is used and any implicit code needs to be generated. This pragma should be used with caution.

F.1.8 PRAGMA initialize

PRAGMA initialize(static | dynamic)

when placed in a library-level package, spec or body; causes all objects in the package to be initialized as indicated, statically or dynamically. Only library-level objects are subject to static initialization. All objects within procedures are, by definition, dynamic.

If PRAGMA initialize(static) is used and an object cannot be initialized statically, code will be generated to initialize the object and a warning message will be generated.

F.1.9 PRAGMA inline

PRAGMA inline

is implemented as described in Appendix B of the Ada RM with the addition that recursive calls can be expanded with the pragma up to the maximum depth of 4. Warnings are produced for bodies that are not available for inline expansion. PRAGMA inline is ignored and a warning is issued when it is applied to subprograms which declare tasks, packages, exceptions, types or nested subprograms.

F.1.10 PRAGMA inline_only

PRAGMA inline_only

when used in the same way as PRAGMA inline, indicates to the compiler that the subprogram must always be in-lined (very important for some code procedures.). This pragma also suppresses the generation of a callable version of the routine which saves code space. If a user erroneously makes an INLINE_ONLY subprogram recursive, a warning message will be emitted and a PROGRAM_ERROR will be raised at run time.

F.1.11 PRAGMA interface

PRAGMA interface (language, subprogram)

supports calls to ADA, C, PASCAL, FORTRAN and UNCHECKED language functions. The Ada specifications can be either functions or procedures. This pragma can also be used to call code written in unspecified languages using UNCHECKED for the language name.

For ADA, the compiler will generate the call as if it were to an Ada procedure but will not expect a matching procedure body.

For C, the types of parameters and the result type for functions must be scalar, access or the predefined type ADDRESS in SYSTEM.ADDRESS. Record and array objects can be passed by reference using the 'ADDRESS attribute. All parameters must have mode IN.

For PASCAL, the types of parameters and the result type for functions must be scalar, access or the predefined type ADDRESS in SYSTEM.ADDRESS. Record and array objects can be passed by reference using the ADDRESS attribute.

For FORTRAN, all parameters are passed by reference; the parameter types must have type SYSTEM.ADDRESS. The result type for a FORTRAN function must be a scalar type.

UNCHECKED may be used to interface to an unspecified language, such as assembler. The compiler will generate the call as if it were to an Ada procedure but will not expect a matching Ada procedure body.

F.1.12 PRAGMA interface_name

PRAGMA interface_name(ada_name, link_name)

with the parameters allows variables or subprograms defined in another language to be referenced directly in Ada. It replaces all occurrences of Ada_name with an external reference to link_name in the object file using the syntax:

PRAGMA interface_name (Ada_name, link_name);

If Ada_name denotes an object, the pragma is allowed at the place of a declarative item in a package specification and must apply to an object declared earlier in the same package specification. The object must be declared as a scalar or an access type. The object cannot be any of the following.

- loop variable
- constant
- initialized variable
- array
- record

If Ada_name denotes a subprogram, a PRAGMA interface must have already been specified for the subprogram.

The link_name must be constructed as expected by the linker. For example, some C compilers and linkers preface the C variable name with an underscore. Such conventions are defined in package LANGUAGE. The following example makes the C global variable errno available within an Ada program:

```
with LANGUAGE;
package PACKAGE_NAME is
        ...
    ERRNO:INTEGER;
    PRAGMA interface_name (ERRNO,LANGUAGE.C_PREFIX & "errno");
    ...
end PACKAGE_NAME;
```

F.1.13 PRAGMA link_with

PRAGMA link_with

can be used to pass arguments to the target linker. It may appear in any declarative part and accepts one argument, a constant string expression. This argument is passed to the target linker whenever the unit containing the pragma is included in a link.

If the constant string expression begins with "-", the string is left untouched.

F.1.14 PRAGMA list

PRAGMA list

is implemented as described in Appendix B of the Ada RM.

F.1.15 PRAGMA memory_size

PRAGMA memory_size

is recognized by the implementation but has no effect in the current release. The implementation does not allow package SYSTEM to be modified by means of pragmas; it must be recompiled.

F.1.16 PRAGMA no_image

PRAGMA no_image

suppresses the generation of the image array used for the IMAGE attribute of enumeration types. This eliminates the overhead required to store the array in the executable image. An attempt to use the IMAGE attribute on a type whose image array has been suppressed will result in a compilation warning and PROGRAM_ERROR raised at run time.

F.1.17 PRAGMA non_reentrant

PRAGMA non_reentrant(subprogram)

takes one argument which can be the name of a library subprogram or a subprogram declared immediately within a library package specification or body. This pragma indicates to the compiler that the subprogram will not be called recursively allowing the compiler to perform specific optimizations. The pragma can be applied to a subprogram or a set of overloaded subprograms within a package specification or package body.

F.1.18 PRAGMA not_elaborated

PRAGMA not_elaborated

suppresses the generation of elaboration code and issues warnings if elaboration code is required. It indicates that the package will not be elaborated because it is either part of the RTS, a configuration package or an Ada package that is referenced from a language other than Ada. It can appear only in a library package specification.

F.1.19 PRAGMA optimize

PRAGMA optimize

is recognized by the implementation but has no effect in the current release.

F.1.20 PRAGMA optimize_code

PRAGMA optimize_code(off | on)

specifies whether the code should be optimized (ON) by the compiler or not (OFF). It can be used in any subprogram. When OFF is specified, the compiler generates unoptimized code. The default is ON.

Optimization can be selectively suppressed using this pragma at the subprogram level. Inline subprograms are optimized even if they have PRAGMA optimize_code(off) unless the caller also has PRAGMA optimize_code(off).

F.1.21 PRAGMA pack

PRAGMA pack

will cause the compiler to minimize gaps between components in the representation of composite types. Objects larger than a single STORAGE_UNIT are packed to the nearest STORAGE_UNIT.

F.1.22 PRAGMA page

PRAGMA page

is implemented as described in Appendix B of the Ada RM. It is also recognized by the source code formatting tool VADS PR (VMS).

F.1.23 PRAGMA passive

PRAGMA passive is not implemented in the MACS RTE. Refer to the CIFO PRAGMA thread_of_control described in the other portions of the MACS SUM.

F.1.24 PRAGMA priority

PRAGMA priority

is implemented as described in Appendix B of the Ada RM. The allowable range for pragma PRIORITY is 0 .. 99.

F.1.25 PRAGMA rts_interface

PRAGMA rts_interface(rts_routine, user_routine)

allows for the replacement of the default calls made implicitly at run-time to the underlying RTS routines. You can cause the compiler to generate calls to any routine of your choosing as long as its parameters and RETURN value match the original. Use this pragma with caution.

F.1.26 PRAGMA share_code

PRAGMA share_code(generic unit/instantiation,boolean) provides for the sharing of object code between multiple instantiations of the same generic subprogram or package body. A 'parent' instantiation is created and subsequent instantiations of the same types can share the parent's object code, reducing program size and compilation times.

The SHARE_CODE pragma takes the name of a generic instantiation or a generic unit as the first argument and either one of the identifiers TRUE or FALSE as a second argument. When the first argument is a generic unit, the pragma applies to all instantiations of that generic. When the first argument is the name of a generic instantiation, the pragma applies only to the specified instantiation or overloaded instantiations.

If the second argument is TRUE, the compiler will try to share code generated for a generic instantiation with code generated for other instantiations of the same generic. When the second argument is FALSE each instantiation will get a unique copy of the generated code.

The extent to which code is shared between instantiations depends on this pragma and the kind of generic formal parameters declared for the generic unit. It is only allowed immediately at the place of a declarative item in a declarative part or package specification or after a library unit in a compilation but before any subsequent compilation unit.

The name PRAGMA share_body may be used instead of share_code with the same effect.

F.1.27 PRAGMA shared

PRAGMA shared is recognized by the implementation but has no effect in the current release.

F.1.28 PRAGMA storage_unit

PRAGMA storage_unit is recognized by the implementation but has no effect in the current release. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the same effect can be achieved by recompiling package SYSTEM with altered values.

F.1.29 PRAGMA suppress

PRAGMA suppress

is implemented as described in Appendix B of the Ada RM except that DIVISION_CHECK and in some cases OVERFLOW_CHECK, cannot be suppressed.

The use of PRAGMA suppress(all_checks) is equivalent to writing at the same point in the program a suppress pragma for each of the checks listed in RM 11.7.

The pragma SUPPRESS(EXCEPTION_TABLES) informs the code generator that the tables normally generated to identify exception regions are not to be generated for the enclosing compilation unit. This reduces the size of the static data required for a unit but also disables exception handling within that unit.

F.1.30 PRAGMA system_name

PRAGMA system_name

is recognized by the implementation but has no effect in the current release. The implementation does not allow SYSTEM to be modified by means of pragmas. However, the file system.a from the STANDARD library can be copied to a local VADS library and recompiled there with new values.

F.1.31 PRAGMA unchecked_subprogram_invocation

PRAGMA unchecked_subprogram_invocation

is TBD.

F.1.32 PRAGMA volatile

PRAGMA volatile(object)

guarantees that loads and stores to the named object will be performed as expected after optimization.

The object declaration and the pragma must both occur (in this order) immediately within the same declarative part or package specification.

F.1.33 PRAGMA warnings

PRAGMA warnings (on | off)

selectively suppress warnings on a single statement or a group of statements.

PRAGMA warnings (off); statement(s) that generate warnings; PRAGMA warnings (on);

F.2 Predefined Packages And Generics

The following predefined Ada packages given by Ada RM Appendix C(22) are provided in the STANDARD library. For VADScross products, they are also provided in the CROSS_IO library.

- generic function UNCHECKED_CONVERSION
- generic package DIRECT_IO
- generic package SEQUENTIAL_IO
- generic procedure UNCHECKED_DEALLOCATION
- package CALENDAR
- package IO_EXCEPTIONS
- package LOW_LEVEL_IO
- package MACHINE_CODE
- package STANDARD
- package SYSTEM
- package TEXT_IO

F.2.1 package SYSTEM

package SYSTEM
The following is the package specification for package SYSTEM:

```
------------------------------------------------------------------
-- |         RESTRICTED RIGHTS LEGEND                            |
-- | Use, duplication, or disclosure by the Government is        |
-- | subject to restrictions as set forth in subparagraph        |
-- | (c)(1)(ii) of the Rights in Technical Data and Computer     |
-- | Software Clause at DFARS 252.227-7013.                      |
-- |         Texas Instruments, Inc.                             |
-- |         6620 Chase Oaks Blvd                                |
-- |         Plano, TX 75023                                     |
```

```
------------------------------------------------------------------
-- | Copyright (c) 1992 Texas Instruments, Inc.                  |
-- | All Rights Reserved.                                        |
------------------------------------------------------------------
with UNSIGNED_TYPES;
package SYSTEM is
pragma SUPPRESS(ALL_CHECKS);
pragma SUPPRESS(EXCEPTION_TABLES);
pragma NOT_ELABORATED;
type NAME is ( MACS );
SYSTEM_NAME : constant NAME := MACS;
STORAGE_UNIT : constant := 8;
MEMORY_SIZE : constant := 16_777_216;

-- System-Dependent Named Numbers
MIN_INT : constant := -2_147_483_648;
MAX_INT : constant := 2_147_483_647;
MAX_DIGITS : constant := 15;
MAX_MANTISSA : constant := 31;
FINE_DELTA : constant := 2.0**(-31);
TICK : constant := 0.01;

-- Other System-dependent Declarations
subtype PRIORITY is INTEGER range 0 .. 99;
MAX_REC_SIZE : integer := 64*1024;
type ADDRESS is private;
function ">" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
function "<" (A: ADDRESS; B: ADDRESS) return BOOLEAN;
function ">="(A: ADDRESS; B: ADDRESS) return BOOLEAN;
function "<="(A: ADDRESS; B: ADDRESS) return BOOLEAN;
function "-" (A: ADDRESS; B: ADDRESS) return INTEGER;
function "+" (A: ADDRESS; I: INTEGER) return ADDRESS;
function "-" (A: ADDRESS; I: INTEGER) return ADDRESS;
function "+" (I: UNSIGNED_TYPES.UNSIGNED_INTEGER) return ADDRESS;
function MEMORY_ADDRESS (I: UNSIGNED_TYPES.UNSIGNED_INTEGER)
                return ADDRESS renames "+";

NO_ADDR : constant ADDRESS;
type TASK_ID is private;
NO_TASK_ID : constant TASK_ID;
subtype SIG_STATUS_T is INTEGER;
SIG_STATUS_SIZE: constant := 4;
type PROGRAM_ID is private;
NO_PROGRAM_ID : constant PROGRAM_ID;
type LONG_ADDRESS is private;
NO_LONG_ADDR : constant LONG_ADDRESS;

function "+" (A: LONG_ADDRESS; I: INTEGER) return LONG_ADDRESS;
```

```
function "-" (A: LONG_ADDRESS; I: INTEGER) return LONG_ADDRESS;
function MAKE_LONG_ADDRESS (A: ADDRESS) return LONG_ADDRESS;
function LOCALIZE(A: LONG_ADDRESS ; BYTE_SIZE : INTEGER)
          return ADDRESS;
function STATION_OF(A: LONG_ADDRESS) return INTEGER;

        -- constant for use in attaching tasks to interrupts
        SBC_NMI_INTERRUPT: constant ADDRESS;

-- Constants describing the configuration of the CIFO add-on product.
SUPPORTS_INVOCATION_BY_ADDRESS : constant BOOLEAN := TRUE;
SUPPORTS_PREELABORATION : constant BOOLEAN := TRUE;
MAKE_ACCESS_SUPPORTED : constant BOOLEAN := TRUE;

-- Arguments to the CIFO pragma INTERRUPT_TASK.
type INTERRUPT_TASK_KIND is ( SIMPLE, SIGNALLING );

        -- Returned by 'task_id for passive tasks.
        type PASSIVE_TASK_ID is private ;
private
type ADDRESS is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
NO_ADDR : constant ADDRESS := 0;
        type PASSIVE_TASK_ID is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
        SBC_NMI_INTERRUPT: constant ADDRESS := 16#4000_0000#;
pragma BUILT_IN(">");
pragma BUILT_IN("<");
pragma BUILT_IN(">=");
pragma BUILT_IN("<=");
pragma BUILT_IN("-");
pragma BUILT_IN("+");
type TASK_ID is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
NO_TASK_ID : constant TASK_ID := 0;
type PROGRAM_ID is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
NO_PROGRAM_ID : constant PROGRAM_ID := 0;
type LONG_ADDRESS is new UNSIGNED_TYPES.UNSIGNED_INTEGER;
NO_LONG_ADDR : constant LONG_ADDRESS := 0;
pragma BUILT_IN(MAKE_LONG_ADDRESS);
pragma BUILT_IN(LOCALIZE);
pragma BUILT_IN(STATION_OF);
end SYSTEM;
```

F.2.2 package CALENDAR

package CALENDAR
The following is the specification of package CALENDAR:

```
package CALENDAR is
    type TIME is private;
```

```
    subtype YEAR_NUMBER   is INTEGER   range 1901 .. 2099;
    subtype MONTH_NUMBER  is INTEGER   range 1 .. 12;
    subtype DAY_NUMBER    is INTEGER   range 1 .. 31;
    subtype DAY_DURATION  is DURATION  range 0.0 .. 86_400.0;

    function CLOCK return TIME;

    function YEAR   (DATE : TIME) return YEAR_NUMBER;
    function MONTH  (DATE : TIME) return MONTH_NUMBER;
    function DAY    (DATE : TIME) return DAY_NUMBER;
    function SECONDS(DATE : TIME) return DAY_DURATION;

    procedure SPLIT (DATE    : in  TIME;
                     YEAR    : out YEAR_NUMBER;
                     MONTH   : out MONTH_NUMBER;
                     DAY     : out DAY_NUMBER;
                     SECONDS : out DAY_DURATION);

    function TIME_OF(YEAR    : YEAR_NUMBER;
                     MONTH   : MONTH_NUMBER;
                     DAY     : DAY_NUMBER;
                     SECONDS : DAY_DURATION := 0.0) return TIME;

    function "+"    (LEFT : TIME;     RIGHT : DURATION) return TIME;
    function "+"    (LEFT : DURATION; RIGHT : TIME)     return TIME;
    function "-"    (LEFT : TIME;     RIGHT : DURATION) return TIME;
    function "-"    (LEFT : TIME;     RIGHT : TIME)     return DURATION;

    function "<"    (LEFT, RIGHT : TIME) return BOOLEAN;
    function "<="   (LEFT, RIGHT : TIME) return BOOLEAN;
    function ">"    (LEFT, RIGHT : TIME) return BOOLEAN;
    function ">="   (LEFT, RIGHT : TIME) return BOOLEAN;

    TIME_ERROR : exception;   --  can be raised by TIME_OF, "+", and "-"

private

    type TIME is record
        MSH : Integer;
        LSH : Integer;
    end record;

end;
```

F.2.3 package MACHINE_CODE

Package MACHINE_CODE provides an assembly language interface for the target machine including the necessary record types needed in the code statement, an enumeration type containing all the opcode mnemonics, a set of register definitions, and a set of addressing mode functions. Also supplied (for use only in units that WITH MACHINE_CODE) are PRAGMA implicit_code and the attribute 'REF.
Machine code statements take operands of type OPERAND, a private type that forms the basis of all machine code address formats for the target.
The general syntax of a machine code statement is

```
CODE_n'(opcode, operand [, operand]));
```

where n indicates the number of operands in the aggregate.
When there is a variable number of operands, they are listed within a subaggregate using this syntax:

```
CODE_n'(opcode, (operand [, operand]);
```

For those opcodes requiring no operands, named notation must be used.

```
CODE_0'(op => opcode);
```

The opcode must be an enumeration literal (i.e., it cannot be an object, attribute or a rename). An operand can only be an entity defined in MACHINE_CODE or the 'REF attribute.
The 'REF attribute denotes the effective address of the first of the storage units allocated to the object. 'REF is not supported for a package, task unit or entry.
Arguments to any of the functions defined in MACHINE_CODE must be static expressions, string literals or the functions defined in MACHINE_CODE.

F.2.4 package SEQUENTIAL_IO

Sequential I/O is not supported by the RTE.

F.2.5 package UNSIGNED_TYPES

package UNSIGNED_TYPES is supplied to illustrate the definition of and services for the unsigned types supplied in this version of VADS. Rational Software Corporation does not give any warranty, expressed or implied, for the effectiveness or legality of this package. It can be used at your own risk.
Rational Software Corporation intends to withdraw this implementation if and when the AJPO and the Ada community reach agreement on a practical unsigned types specification. We can then standardize on that accepted version at a practical date thereafter.
The package is supplied in comment form because the actual package cannot be expressed in normal Ada - the types are not symmetric about 0 as required by the Ada RM. This package is supplied and is accessible through the Ada WITHn statement as though it were present in source form.
Example:

```
with unsigned_types;
procedure foo( xxx: unsigned_types.unsigned_integer) is ...
```

CAUTION: Use package UNSIGNED_TYPES at your own risk.
A complete specification of package UNSIGNED_TYPES can be found in Appendix F of the VADS Programmer's Guide.

F.3 Slices

A slice denotes a one-dimensional array formed by a sequence of consecutive components of a one-dimensional array. A slice of a variable is a variable; a slice of a constant is a constant; a slice of a value is a value. The syntax is:

prefix(discrete_range)

The prefix of a slice must be appropriate for a one-dimensional array type. The type of the slice is the base type of this array type. The bounds of the discrete range define those of the slice and must be of the type of the index; the slice is a null slice denoting a null array if the discrete range is a null range.
For the evaluation of a name that is a slice, the prefix and the discrete range are evaluated in some order that is not defined by the language. The exception CONSTRAINT_ERROR is raised by the evaluation of a slice, other than a null slice, if any of the bounds of the discrete range does not belong to the index range of the prefixing array. (The bounds of a null slice need not belong to the subtype of the index.)

F.4 Implementation-defined Attributes

'REF

The 'REF attribute denotes the effective address of the first of the storage units allocated to the object. 'REF is not supported for a package, task unit or entry. There are two forms of use for this attribute, X'REF and SYSTEM.ADDRESS'REF(N). X'REF is used only in machine code procedures while SYSTEM.ADDRESS'REF(N) can be used anywhere to convert an integer expression to an address.

X'REF

The attribute generates a reference to the entity to which it is applied.
In X'REF, X must be either a constant, variable, procedure, function or label. The attribute returns a value of the type MACHINE_CODE.OPERAND and may only be used to designate an operand within a code-statement.
The instruction generated by the code-statement in which the attribute occurs may be preceded by additional instructions needed to facilitate the reference (i.e., loading a base register). If the declarative section of the procedure contains PRAGMA implicit_code (off), a warning will be generated if additional code is required.

SYSTEM.ADDRESS'REF(N)

The effect of this attribute is similar to the effect of an unchecked conversion from integer to address. However, SYSTEM.ADDRESS'REF(N) should be used instead in the following listed circumstances and in these circumstances, N must be static.

In SYSTEM.ADDRESS'REF(N), N must be an expression of type UNIVERSAL_INTEGER and for all products but VADSself on VAX VMS, SYSTEM.ADDRESS must be the type SYSTEM.ADDRESS. The attribute returns a value of type SYSTEM.ADDRESS, which represents the address designated by N.

Within any of the run time configuration packages:

Use of unchecked conversion within an address clause would require the generation of elaboration code but the configuration packages are not elaborated.

In any instance where N is greater than INTEGER'LAST:

Such values are required in address clauses which reference the upper portion of memory. To use unchecked conversion in these instances would require that the expression be given as a negative integer.

To place an object at an address, use the 'REF attribute:

The integer_value in the following example is converted to an address for use in the address representation clause. The form avoids UNCHECKED_CONVERSION and is also useful for 32-bit unsigned addresses.

```
--place an object at an address
for object use at ADDRESS'REF (integer_value)

--to use unsigned addresses
for VECTOR use at SYSTEM.ADDRESS'REF(16#808000d0#);
TOP\_OF\_MEMORY : SYSTEM.ADDRESS :=
                    SYSTEM.ADDRESS'REF(16#FFFFFFFF#);
```

X'TASK_ID

For a task object or a value, X, X'TASK_ID yields the unique task ID associated with the task. The value of this attribute is of the type SYSTEM.TASK_ID.

F.5 Restrictions On 'Main' Programs

VADS requires that a 'main' program must be a non-generic subprogram that is either a procedure or a function returning an Ada STANDARD.INTEGER (the predefined type). A 'main' program may be neither a generic subprogram nor an instantiation of a generic subprogram.

F.6 Generic Declarations

VADS does not require that a generic declaration and the corresponding body be part of the same compilation and they are not required to exist in the same VADS library. An error is generated if a single compilation contains two versions of the same unit.

F.6.1 Shared Object-code For Generic Subprograms

The VADS compiler generates code for a generic instantiation that can be shared by other instantiations of the same generic thus reducing the size of the generated code and increasing compilation speed. There is an overhead associated with the use of shared code instantiations because the generic actual parameters must be accessed indirectly and in the case of a generic package instantiation, declarations in the package are also accessed indirectly. Also, greater optimization is possible for unshared instantiations because exact actual parameters are known. It is the responsibility of the programmer to decide whether space or time is most critical in a specific application.
To give the programmer control of when an instantiation generates unique code or shares code with other similar instantiations, PRAGMA share_code is provided. This PRAGMA can be applied to a generic declaration or to individual instantiations.
It is not practical to share the code for instantiations of all generics. If the generic has a formal private type parameter the generated code to accommodate an instantiation with an arbitrary actual type would be extremely inefficient.
The VADS compiler does not share code by default. The INFO directive SHARE_BODY may be specified in a VADS library to cause the compiler to always share generic code bodies. PRAGMA share_code may be applied to generic units or generic instances to control whether specific instances are shared.
To override the default, the PRAGMA share_code(name, false) must be used. If there are formal subprogram parameters instantiations will not be shared unless an explicit PRAGMA share_code(name, true) is used.
The PRAGMA share_code is used to indicate desire to share or not share an instantiation. The PRAGMA can reference either the generic unit or the instantiated unit. When it references a generic unit, it sets sharing on or off for all instantiations of that generic unless overridden by specific share_code PRAGMAS for individual instantiations. When it references an instantiated unit, sharing is on or off only for that unit.
The PRAGMA share_code is only allowed in the following places: immediately within a declarative part, immediately within a package specification or after a library unit in a compilation but before any subsequent compilation unit. The form of this PRAGMA is

```
PRAGMA share_code (generic, boolean_literal)
```

Note that a parent instantiation (the instantiation that creates the sharable body) is independent of any individual instantiation, therefore reinstantiation of a generic with different parameters has no effect on other compilations that reference it. The unit that caused compilation of a parent instantiation need not be referenced in any way by subsequent units that share the parent instantiation.
Sharing generics causes a slight execution time penalty because all type attributes must be indirectly referenced (as if an extra calling argument were added). However, it substantially reduces compilation time in most circumstances and reduces program size.

We have compiled a unit, SHARED_IO, in the standard library that instantiates all Ada generic I/O packages for the most commonly used base types. Thus, any instantiation of an Ada I/O generic package will share one of the parent instantiation generic bodies unless the following PRAGMA is used:

```
PRAGMA share_code ( generic, false );
```

F.7 Representation Clauses

F.7.1 Bit-level Representation Clauses

VADS supports bit-level representation clauses.

F.7.2 PRAGMA pack

VADS does not define any additional representation PRAGMAS.

F.7.3 Length Clauses

VADS supports all representation clauses.

F.7.4 Enumeration Representation Clauses

Enumeration representation clauses are supported.

F.7.5 Record Representation Clauses

Representation clauses are based on the target machine's word, byte and bit order numbering so that VADS is consistent with machine architecture manuals for both 'big-endian' and 'little-endian' machines. Bits within a STORAGE_UNIT are also numbered according to the target machine manuals. It is not necessary for a user to understand the default layout for records and other aggregates since fine control over the layout is obtained by the use of record representation clauses. It is then possible to align record fields correctly with structures and other aggregates from other languages by specifying the location of each element explicitly. The 'FIRST_BIT and 'LAST_BIT attributes can be used to construct bit manipulation code that is applicable to differently bit-numbered systems.
A figure (or figures) illustrating the addressing and bit numbering scheme available for your target system can be found in Appendix F of the Programmer's Guide.
The only restriction on record representation clauses is that if a component does not start and end on a storage unit boundary, it must be possible to get it into a register with one move instruction.

F.8 Change of Representation

Change of representation is supported.

F.9 package SYSTEM

The specification of package SYSTEM is available in the Appendix F chapter of the Programmer's Guide. The PRAGMAs system_name, storage_unit and memory_size are recognized by the implementation but have no effect. The implementation does not allow SYSTEM to be modified by means of PRAGMAS. however, the same effect can be achieved by recompiling the SYSTEM package with altered values. Note that such a compilation will cause other units in the STANDARD library to become out of date. Consequently, such recompilations should be made in a library other than standard.

F.9.1 System-Dependent Named Numbers

The specification of package SYSTEM is listed on page F.2.1. This specification is also available on line in the file system.a in the release standard library.

F.10 Representation Attributes

F.10.1 'ADDRESS attribute

The 'ADDRESS attribute is supported for the following entities.

- variables
- constants
- procedures
- functions

If the prefix of an address attribute is an object that is not aligned on a storage unit boundary, the attribute yields the address of the storage unit containing the first bit of the object. This is consistent with the definition of the FIRST_BIT attribute.
All other Ada representation attributes are fully supported.

F.10.2 Representation Attributes of Real Types

These attributes are supported.

F.11 Machine Code Insertions

Machine code insertions are supported.

F.12 Interface to Other Languages

The VADS interface to other languages is discussed in the Interface Programming chapter in the VADS Reference Guide and in the section PRAGMAS and Their Effects.

F.13 Unchecked Programming

Both UNCHECKED_DEALLOCATION and UNCHECKED_CONVERSION are provided.

F.13.1 Unchecked Storage Deallocations

Any object that was allocated may be deallocated. When an object is deallocated, its access variable is set to null. Subsequent deallocations using the null access variable are ignored.

F.13.2 Unchecked Type Conversions

The predefined generic function UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.

F.14 Parameter Passing

Parameters are passed in registers and/or by pushing values (or addresses) on the stack. Extra information is passed for records ('CONSTRAINED) and for arrays (dope vector address).
Small results are returned in registers; large results with known targets are passed by reference. Large results of anonymous target and known size are passed by reference to a temporary created on the caller's stack. Large results of anonymous target and unknown size are returned by copying the value down from a temporary in the callee so that the space used by the temporary can be reclaimed.
The compiler assumes the following calling conventions.

1. Caller passes scalar arguments in a0, a1, a2 and a3 and floating pointer arguments in f12 and f14. Other arguments are passed on the stack. Inter-language calls (for example, from a C routine to an Ada routine or from an Ada routine to a C routine) use the standard MIPS calling convention. To call a C procedure, declare the Ada interface using PRAGMA interface (language, subprogram). To declare an Ada procedure that will be called from C or FORTRAN, use PRAGMA external (language, subprogram).

2. Caller calls callee.

3. Callee allocates space for locals, if needed, by subtracting from the stack pointer. If the stack pointer is changed, then a stack overflow check is executed.

4. Callee preserves registers in the set s0-s7, sp, s8 and ra if they are used. Also, registers f20 through f30 are saved if used.

5. Callee copies the display, if needed.

6. Callee sets up a frame pointer (r30, aka fp) if the sp is modified by code during the call. Otherwise, a virtual frame pointer is used.

7. Callee executes.

8. Callee puts return result in v0 or f0.

9. Saved registers are restored.

10. The stack pointer is restored, which reclaims local storage.

11. The callee returns to the caller.

Note that machine code insertions can be used to explicitly build a call interface when compiler conventions are not compatible or when interfacing to assembly language.
It is important to understand the referencing of parameters when using machine_code insertions. Parameters cannot be treated like memory locations since in may cases, they are being held in registers. Attempting to treat a parameter held in a register like a memory location will cause a compiler error.

F.15 Conversion And Deallocation

The predefined generic function UNCHECKED_CONVERSION cannot be instantiated with a target type that is an unconstrained array type or an unconstrained record type with discriminants.
There are no restrictions on the types with which generic function UNCHECKED_DEALLOCATION can be instantiated. No checks are performed on released objects.

F.16 Types, Ranges and Attributes

The maximum ARRAY, RECORD and TYPE size limits have been increased to 256_000_000.

F.17 Numeric Literals

VADS Ada uses unlimited precision arithmetic for computations with numeric literals.

F.18 Enumeration Types

VADS Ada allows an unlimited number of literals within an enumeration type.

F.19 Attributes of Discrete Types

VADS Ada defines the image of a character that is not a graphic character as the corresponding 2- or 3-character identifier from package ASCII of Ada RM Annex C-4. The identifier is in upper case without enclosing apostrophes. For example, the image for a carriage return is the 2-character sequence CR (ASCII.CR).

F.20 The type STRING

Except for memory size, VADS Ada places no specific limit on the length of the predefined type STRING. Any type derived from the type STRING is similarly unlimited.

F.21 The type INTEGER

The following are the INTEGER attribute values:

type INTEGER Attribute Values			
Name of Attribute	Attribute Value of INTEGER	Attribute Value of SHORT_INTEGER	Attribute Value of TINY_INTEGER
SIZE	32	16	8
FIRST	-2_147_483_648	-32_768	-128
LAST	2_147_483_647	32_767	127

F.22 Operation of Floating Point Types

VADS Ada floating point types have the attributes given in the following table:

Floating Point Types		
Name of Attribute	Attribute Value of LONG_FLOAT	Attribute Value of FLOAT SHORT_FLOAT
SIZE	64	32
FIRST	-1.79769313486232E+308	-3.40282E+38
LAST	1.79769313486232E+308	3.40282E+38
DIGITS	15	6
MANTISSA	51	21
EPSILON	8.88178419700125E-16	9.53674316406250E-07
EMAX	204	84
SMALL	1.94469227433161E-62	2.58493941422821E-26
LARGE	2.57110087081438E+61	1.93428038904620E+25
SAFE_EMAX	1021	125
SAFE_SMALL	2.22507385850720E-308	1.17549435082229E-38
SAFE_LARGE	2.24711641857789E+307	4.25352755827077E+37
MACHINE_RADIX	2	2
MACHINE_MANTISSA	53	24
MACHINE_EMAX	1024	128
MACHINE_EMIN	-1021	-125
MACHINE_ROUNDS	TRUE	TRUE
MACHINE_OVERFLOWS	TRUE	TRUE

F.23 Fixed Point Type Attribute Values

VADS Ada fixed point types have the attributes given in the following table:

Fixed Point Type Attribute Values	
Name of Attribute	Attribute Value for DURATION
SIZE	32
FIRST	-214748.3648
LAST	214748.3647
DELTA	1.00000000000000E-04
MANTISSA	31
SMALL	1.00000000000000E-04
LARGE	2.14748364700000E+05
FORE	7
AFT	4
SAFE_SMALL	1.00000000000000E-04
SAFE_LARGE	2.14748364700000E+05
MACHINE_ROUNDS	TRUE
MACHINE_OVERFLOWS	TRUE